虚拟化

线程相关的状态机视角

- fork：对当前状态机状态进行完整复制
- execve：将当前状态机状态重置为某个可执行文件描述的状态机
- exit：销毁当前状态机

程序 = 状态机
执行指令的机器，计算指令，系统调用指令
操作系统，特殊的状态机，管理状态机的状态机。

Fork bomb

```
pid_t x = fork();
pid_t y = fork();
printf("%d %d\n", x, y);
```

阅读程序，写出运行结果

```
for (int i = 0; i < 2; i++) {
  fork();
  printf("Hello\n");
}
```

状态机视角帮助我们严格理解程序行为

- ./a.out
- ./a.out | cat
  - 计算机系统里没有魔法
  - (无情执行指令的) 机器永远是对的

上面是6，下面是8，因为printf会因为输出对象的不同而改变
前者是输出到终端，后者是缓冲区，fork会复制缓冲区哦。

execve 创建状态机
这个会继承fd，env

exit（）：C给你提供的，会调用 atexit
_exit()：glibc 的 syscall wrapper，执行 “exit_group” 系统调用
终止整个进程 (所有线程)
syscall(SYS_exit, 0)：执行 “exit” 系统调用终止当前线程（不会把这组的线程全部终止），不会调用 atexit

计算机系统里面没有东西是你搞不定的，如果搞不定，就先搞一个简单的，弄清楚他的原理，然后复杂就是在简单的上面再加点东西。
strace
系统里面的工具没什么复杂的，所有工具都是由你提供的。

```
00401000-00498000 r-x 00001000 08:12 9437238      /tmp/demo/a.out (code)
00498000-004c1000 r-- 00098000 08:12 9437238      /tmp/demo/a.out (rodata)
004c1000-004c8000 rw- 000c0000 08:12 9437238      /tmp/demo/a.out (data)
004c8000-004cd000 rw- 00000000 00:00 0            [heap]
7ffff7ff9000-7ffff7ffd000 r-- 00000000 00:00 0    [vvar]
7ffff7ffd000-7ffff7fff000 r-x 00000000 00:00 0    [vdso]
7ffffffde000-7fffffffff000 rw- 00000000 00:00 0   [stack]
ffffffffff600000-ffffffffff601000 --x 00000000    [vsyscall]
```

everything is file
so fd is ed

系统调用和shell

防御性编程，调试理论，与吃苦和熬夜

所有的程序，都是用户需求在程序世界的投影

shell程序，语法树
任何编程语言，都是字符串集的预编译
shell online judge，加快速度，减少tedious

C标准库
Freestanding

memset是不上锁的，为了性能。
首先是上锁很困难，然后是你用户这么搞，之后应该有对应的操作为此负责。

lib.c
里面的文件描述符对操作系统的封装。
环境变量
的赋值?
环境变量是放在argc和argv后面的，
环境变量的指针，一开始的时候对应的值是0，然后就在函数初始化的时候传进去
OS会帮你传入对应参数

对象和环境的封装。

在考虑性能的优化之前，你要知道什么是性能。
OS的数据结构和数据结构课里面的数据结构是不一样的

实现高效的 malloc/free

> Premature optimization is the root of all evil.
> ——D. E. Knuth

重要的事情说三遍：

- 脱离 workload 做优化就是耍流氓
- 脱离 workload 做优化就是耍流氓
- 脱离 workload 做优化就是耍流氓
  - 在开始考虑性能之前，理解你需要考虑什么样的性能

然后，去哪里找 workload?

- 当然是 paper 了 (顺便白得一个方案)
  - Mimalloc: free list sharding in action (APLAS'19)

Workload 分析

在实际系统中，我们通常不考虑 adversarial 的 worst case。

- 指导思想：$O(n)$ 大小的对象分配后至少有 $\Omega(n)$ 的读写操作，否则就是 performance bug (不应该分配那么多)
  - 部小的对象创建/分配越频繁
    - 字符串、临时对象等；生存周期可长可短
  - 较为频繁地分配中等大小的对象
    - 较大的数组、复杂的对象；更长的生存周期
  - 低频率的大对象
    - 巨大的容器、分配器；很长的生存周期
- 并行，并行，再并行
  - 所有分配都会在所有处理器上发生
  - 使用链表/区间树 (first fit) 可不是个好想法

Linux （鸡汤）

努力去成为一个能够做出一些东西的人，孩子，不要怕困难，不要害怕权威。
你能想象21岁的年轻人和已经是操作系统神牛的权威打辩论吗?
虽然linux是时代的产物，你不需要发明linux，但是你可以成为自己的linus。他自傲，苛责，追求完美。永远对屎山代码不满意。

对应的课程大概是real world 的引导程序=_=
TODO：之后重看

VM 一些primitive 在用户级进程的一些应用，核心就是**用VM机制，复用了VM的硬件，减少了对应的检查指令。**
交给用户来进行一些关于page fault的处理，给用户自己page fault的机会。（安全性是有保证的，因为这个处理程序是在用户空间里面进行的，最多破坏自己。）

- GC

【GC的本质应该是用类似于BFS，生成一个可达图，然后去掉不可达的东西（没有指针引用的，就无法访问，为garbage。）】

- 一般的copying算法。

具体思路就是，区域分为from和to，然后from区域满后，从根节点开始BFS，把相关的全部放到to里面（过程是，先把根节点放到to，然后用forward这个操作，标记已经放到to的from里面的指针（表现为指针指的是to结点的。）放到to区域的维持的指针仍然是指向from的，只有对应from的指针放到to区域后，才指向to）（根节点是从存放在栈上的和寄存器里面的指针开始建立的。）

- 优化的baker算法，（incremental，copying）

baker是把这个操作分摊到每个操作中，每个操作，自己都会搬运一点from节点的东西到to节点里面。其他都是一样的。到最后也是from区域满了，然后貌似可以直接clean。
vm的引入，去掉了额外的指令（但是某种意义上全部放到了handler里面）同时打包给handler之后，可以很好的处理并发。
在checking point，以及shared memory这种没有编译器能够辅助的，效果不错，这是一个很不错的思路。

Q：你应该在这里使用虚拟内存吗？或者说这里的这些技巧值得吗？
许多的GC并没有使用虚拟内存，而是通过编译器生成的代码来完成GC，并且还有各种其他的技巧来减少性能损耗。所以GC的大部分场景都可以通过一些额外的指令来完成。这对于一个编译器，程序运行时，或者编程语言来说，并不是一个太糟糕的选择，因为编译器就可以完成这些操作。但是如果没有程序运行时或者编译器，那么这个过程就会很痛苦。所以对于一些完全没有编译器参与的应用程序，例如checkpointing，shared-virtual memory，它们的确需要这里提到的虚拟内存特性。实际中，足够多的应用程序开发人员发现了这些特性的价值，所以今天的操作系统都支持了这些虚拟内存特性。

设计空间

- 建模 (理解和总结 “过去发生了什么”)
- 预测 (试图预知未来可能发生什么)
- 决策 (在预测下作出对系统最有利的选择)

允许用户定制处理器调度

UNIX niceness

- -20 .. 19 的整数，越 nice 越让别人得到 CPU
  - -20: 极坏; most favorable to the process
  - 19: 极好; least favorable to the process
- 基于优先级的调度策略
  - RTOS: 坏人躺下好人才能上
    - ~~好人流下了悔恨的泪水~~
  - Linux: 10 nice ≈ CPU 资源获得率相差 10 倍

策略：Complete Fair Scheduling (CFS)

试图去模拟一个 “Ideal Multi-Tasking CPU”:

- “Ideal multi-tasking CPU” is a (non-existent :-)) CPU that has 100% physical power and which can run each task at precise equal speed, in parallel, each at . For example: if there are 2 tasks running, then it runs each at 50% physical power — i.e., actually in parallel.

“让系统里的所有进程尽可能公平地共享处理器”

- 为每个进程记录精确的运行时间
- 中断/异常发生后，切换到运行时间最少的进程执行
  - 下次中断/异常后，当前进程的可能就不是最小的了

CFS: 实现优先级

操作系统具有对物理时钟的 “绝对控制”

- 每人执行 1ms，但好人的钟快一些，坏人的钟慢一些
  - vruntime (virtual runtime)
  - vrt[i] / vrt[j] 的增加比例 = wt[j] / wt[i]

```
const int sched_prio_to_weight[40] = {
 /* -20 */ 88761, 71755, 56483, 46273, 36291,
 /* -15 */ 29154, 23254, 18705, 14949, 11916,
 /* -10 */  9548,  7620,  6100,  4904,  3906,
 /*  -5 */  3121,  2501,  1991,  1586,  1277,
 /*   0 */  1024,   820,   655,   526,   423,
 /*   5 */   335,   272,   215,   172,   137,
 /*  10 */   110,    87,    70,    56,    45,
 /*  15 */    36,    29,    23,    18,    15,
};
```

这个是一个指数函数演化过来的。

- **调度的困难**
- 互斥锁导致优先级变化
- 多处理器导致跨核调度成本极高
- 多用户多任务的时候，多任务的抢少任务的
  - 无法自己随便提高自己的优先级
  - OS套OS导致各种bug
- 电路效能比（功率无法支持所有电路同时工作）CPU频率不固定
- Non-Uniform Memory Access
- 处理器不是越多越快（放弃一半处理器上升速度。）

不要高兴得太早

```
void jyy() { // 最低优先级
  mutex_lock(&wc_lock);
  // 先到先得
}

void xi_zhu_ren() { // 中优先级
  while (1) ;
}

void xiao_zhang() { // 高优先级
  sleep(1);
  mutex_lock(&wc_lock);
  ...
}
```

jyy 在持有互斥锁的时候被赶下了处理器……
互斥锁的引入，导致了有些进程被赶走以后，其他都动不了，也许我们可以暂时给他提供一些更高的优先级。

解决优先级反转问题

Linux: 解决不了，CFS 凑合用吧
实时系统：火星车在 CPU Reset，不能摆烂啊

- 优先级继承 (Priority Inheritance)/优先级提升 (Priority Ceiling)
  - 持有 mutex 的线程/进程会继承 block 在该 mutex 上进程的最高优先级
  - 但也不是万能的 (例如条件变量唤醒)
- 在系统中动态维护资源依赖关系
  - 优先级继承是它的特例
  - 似乎更困难了……
- 避免高/低优先级的任务争抢资源
  - 对潜在的优先级反转进行预警 (lockdep)
  - TX-based: 冲突的 TX 发生时，总是低优先级的 abort

多处理器调度的困难所在

既不能简单地 “分配线程到处理器”

- 线程退出，瞬间处理器开始围观

也不能简单地 “谁空丢给谁”

- 在处理器之间迁移会导致 cache/TLB 全都白给

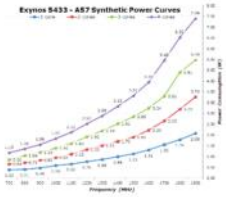
多处理器调度的两难境地

- 迁移？可能过一会儿还得移回来
- 不迁移？造成处理器的浪费

实际情况 (2): Big.LITTLE/能耗 (cont'd)

软件可以配置 CPU 的工作模式

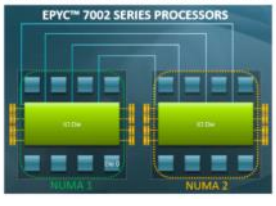
- 开/关/工作频率 (频率越低，能效越好)
- 如何在给定功率下平衡延迟 v.s. 吞吐量?



实际情况 (3): Non-Uniform Memory Access

共享内存只是假象

- L1 Cache 花了巨大的代价才让你感到内存是共享的
- Producer/Consumer 位于同一个/不同 module 性能差距可能很大



```
$ strace ./a.out 8| vim -
Vim: Reading from stdin...
```

```
jyy@jyy-usb:/tmp/demo$ vim <(ls)
```